Suplemento mensual dedicado a la imagen sintética. Número 17

Rendermanía

La Portada. La batalla prometida
Cómo... Primeros pasos con POV (II)
Foro del Lector. Selección de trabajos y preguntas
En el CD. Las escenas de la batalla y sus ficheros

pcmania

Empezar con POV (parte II)

A pesar de que «POV» tiene una presencia casi constante en Rendermanía, cada vez es mayor el número de lectores que declaran en sus cartas no entender muchos de los artículos dedicados a este programa, e incluso hemos recibido misivas donde se nos pregunta "¿qué es POV?". Esta aparente paradoja tiene una fácil explicación. Hace ya casi año y medio explicamos los conceptos básicos de este raytracer y desde entonces no hemos vuelto a hablar de ellos, con lo cual los lectores más recientes no pueden seguir los artículos más complejos. Por esta razón, este mes continuaremos repasando los temas básicos como preludio a otros asuntos más complejos como la creación de texturas procedurales o la animación.

Hay que aclarar, sin embargo, que los presentes artículos no van a ser una repetición del microcurso que dedicamos a «POV» en los números 0 y 1 de Rendermanía. En aquella ocasión nos centramos en gran medida en las sentencias de creación de objetos del lenguaje escénico de «POV» y en cómo modelar con ellas. Esta vez, por el contrario, obviaremos casi toda esta parte, asumiendo que la mayoría de los usuarios van a trabajar con modelos importados construidos desde modeladores gráficos. Así pues estudia-

remos cómo manipular estos modelos importados desde POV y cómo crear texturas procedurales, emplear los efectos de POV, etc.

Más sobre #OBJECT, #DECLARE E #INCLUDE

El mes anterior estudiamos un sencillo ejemplo en el que se utilizaba la utilidad de conversión 3ds2pov para "traducir" un fichero 3ds al formato de POV. Como ya vimos, la utilidad procesó el archivo con el modelo de «3D Studio» y creó dos ficheros para «POV». La malla con la geometría de la vaca se guardó en el archivo vaca.inc. y otros datos como el color de la vaca, la posición y orientación de la cámara y la iluminación se almacenaron en vaca.pov. Por ello puede decirse que, más que un traductor de objetos, 3ds2pov es un traductor de escenas, ya que, salvo por la aplicación de los bitmaps, esta utilidad traduce todos los elementos de la escena original de «3D Studio» al formato de «POV». Sin embargo, aunque gracias a 3ds2pov podríamos limitarnos después, sin hacer nada más, a ordenar el render desde «POV» y aunque esto mismo es posible también –y de manera más sencilla aún– trabajando con otros modeladores (como Rhino), el caso es que de esta manera no estaríamos aprovechando las posibilidades de «POV». Lo ideal es crear los objetos desde otros modeladores y exportarlos a «POV», desde donde los manipularemos para crear la escena utilizando el lenguaje escénico. Esto resulta menos interactivo que realizar todo el trabajo desde un modelador visual como «Rhino» o «3D Studio», pero también presenta importantes ventajas que aprenderemos con el tiempo y por esta razón habremos de seguir estudiando el lenguaje escénico de POV.

Ahora continuaremos estudiando diversas maneras de manipular objetos con el lenguaje escénico. El mes anterior vimos con el ejemplo de vaca.pov y vaca.inc que podíamos crear una vaca en la escena con la siguiente sentencia del archivo vaca.pov:

```
#include "vaca.inc"
```

Figura 1

```
...
...
#declare colorpiel=texture{
        #declare vari=rand(R)
        pigment { color rgb <.10, 1-(vari/4), .20> }
        }
...
...
#declare head=object{#include "head1.inc"}            // la cabeza del orco
#declare torax=object{#include "torax.inc"}           // su tórax
#declare brazo_derecho=object{#include "brazod.inc"} // su brazo derecho
...
...

// la unión siguiente define al modelo completo del orco
union{
object{head rotate y*45}                              // la cabeza rota 45 grados
object{torax }
object{cinturon pigment{MarronClaro}}
...
...
texture{colorpiel}
translate <Xpos,35,Zpos>                              // todo el conjunto se traslada
}   // aquí termina la unión
...
...
```

Esto generaba una vaca en la posición definida por la malla guardada en el fichero vaca.inc. Para poder crear una segunda vaca o para manipular a una única vaca había que utilizar una sentencia object como la que sigue:

object{#include "vaca.inc" translate<344,0,400>}

Con la sentencia object se crea una especie de contenedor definido por los corchetes. Así, las operaciones de transformación espacial que se definan dentro de estos corchetes afectarán únicamente al objeto contenido por ellos. Este detalle –tan parecido a los bloques de C– es una parte fundamental de la filosofía de POV y habremos de tenerlo en cuenta en la manipulación de cualquier objeto compuesto por otros más simples (como es el caso del orco o del guerrero humano). Examinemos ahora el ejemplo de la **Figura 1** (en la página anterior).

En este ejemplo podemos comprobar muchos de los detalles que conforman la filosofía de «POV». Se trata de un extracto de un fichero para crear una figura antropomórfica. La primera sentencia es un #declare con el que se define el identificador "colorpiel" como una textura (que luego podría ser aplicada a un objeto). Después se usan tres #declares para definir tres objetos llamados "head", "torax" y "brazo_derecho". Así pues, como podéis ver, podemos usar la sentencia #declare no sólo para asignar valores a variables sino también para declarar nuevas texturas y objetos. La sintaxis para declarar un objeto es:

#declare identificador=object {sentencias de definición}

Luego, para utilizar lo declarado, tendremos que usar una línea como:

object{identificador sentencias}

Con #declare el objeto definido podrá ser invocado cuantas veces queramos en la escena usando sentencias object. Seguramente el lector se preguntará por qué tenemos que molestarnos en hacer esto cuando podríamos escribir algo como:

object{#include "objeto.inc" sentencias}

...cada vez que deseáramos situar un objeto (guardado en otro fichero) en la escena. Para responder a esto imaginad que queremos situar 200 orcos en nuestra escena (por ejemplo para una batalla). Con #include, «POV» tendría que leer 200 veces el fichero objeto.inc, lo cual sería horriblemente lento (sobre todo si objeto.inc es un modelo poligonal complejo). En cambio usando #declare «POV» sólo leería el fichero una vez, guardaría un modelo en memoria asignándolo al identificador, y lo emplearía cada vez que el identificador fuese invocado con object. Es de suponer que la cantidad total de RAM empleada será similar en ambos casos pero #declare es más cómodo y más versátil (como ve-

Figura 2

```
...
...
#declare Here=<1,2,3>
#declare Count=0                          // initialize Count
union {
   object { Rod translate Here*Count }
   #declare Count=Count+1                 // re-declare inside union
   object { Rod translate Here*Count }
   #declare Count=Count+1                 // re-declare inside union
   object { Rod translate Here*Count }
 }
...
...
```

Las texturas procedurales son tridimensionales, como puede apreciarse en estas imágenes de ejemplo.

remos mas adelante). Aunque #declare nos permite declarar mas cosas además de variables, objetos y texturas, por el momento nos contentaremos con esto.

Un identificador puede tener hasta 40 caracteres y puede emplear letras y números (aunque no debe comenzar con un número). También es importante recordar que «POV» diferencia entre mayúsculas y minúsculas por lo que el identificador "YuYu" no es lo mismo que "yuyu" y también habremos de tener cuidado de no usar como identificador ninguna palabra reservada del lenguaje escénico como sphere, box u otras similares (consultad la lista en povray.doc, en el apartado "Identifiers and Keywords"). Hay que tener cuidado con estos dos últimos detalles.

Las declaraciones pueden colocarse en cualquier punto de los ficheros, incluso dentro de otras sentencias. En la **Figura 2** tenemos un ejemplo tomado de povray.doc

El trabajo con las variables se basa en la facultad que ofrece #declare de emplear un valor asignado anteriormente como parte de una nueva declaración. Sin embargo, hay un caso donde esto no funcionará: si intentamos redifinir un identificador como algo distinto obtendremos un error en el parsing. Así, si por ejemplo hemos definido "hroki" como una textura e intentamos redefinirla después como un objeto, obtendremos un error y lo mismo sucederá si definimos un identificador como un valor e intentamos declararlo luego como un vector.

Uniones

Pero continuemos con el ejemplo de la **Figura 1**. En él veremos que el tercer bloque de sentencias comienza con la palabra unión. Esta sentencia se utiliza para crear objetos compuestos por otros más simples. Su forma de uso es similar al de object, pero con la diferencia de que podemos meter varios objetos dentro del bloque delimitado por los corchetes. Estos objetos no tienen por qué ser objetos simples ya que una unión puede estar compuesta también por otras uniones. La utilidad de las uniones radica en que, a efectos prácticos, «POV» tratará como un objeto único todo lo que se halle dentro de los corchetes de una de estas sentencias. Esto es muy útil porque los modelos complejos raramente suelen estar compuestos por una única pieza. En el caso de nuestro orco, por ejemplo, éste está compuesto por piezas como los brazos, las piernas, la cabeza, etc, cuya posición y orientación nos puede interesar alterar individualmente dentro del conjunto del modelo. Veamos como funciona: la primera sentencia que encontramos dentro del bloque de la unión es

```
object{head rotate y*45}
```

Esta sentencia indica a «POV» que el objeto head ya declarado antes es parte de la unión. Al no haber ninguna orden translate dentro del bloque de este object, «POV» situará a head en la escena en la misma posición en que este objeto se halla definido. (Head, como otros objetos del orco, es un objeto poligonal y su posición inicial viene dada por los valores asignados a los vértices de su malla). Lo que sí tenemos aquí es una transformación rotate que hará girar la cabeza 45 grados. Como la sentencia rotate está encerrada dentro del bloque contenedor de head, esta transformación espacial sólo afectará a la cabeza, siendo ignorada por el resto de los objetos miembros de la unión.

Esta regla de aislamiento de los bloques es simple pero esencial y tenemos otro ejemplo de su funcionamiento en el tercer miembro de la unión donde se asigna el color "MarronClaro" al objeto "cinturón". (La sentencia pigment especifica el color de un objeto y normalmente suele ser parte de la definición de una textura pero, si no vamos a definir otras propiedades para la textura, podemos utilizar pigment fuera de texture). Puesto

que la sentencia pigment está dentro del bloque de "cinturón", el color indicado sólo se aplicará sobre este objeto y no sobre los anteriores.

Por último fijaos en las dos últimas sentencias escritas dentro de la unión. Estas sentencias no agregan nuevos miembros a la unión. La primera es

texture{colorpiel}

que aplica una textura ya declarada anteriormente. Pero, ¿dónde se aplica esta textura? Siguiendo las reglas de los corchetes podemos adivinar que se aplica dentro de la unión, es decir, sobre todos los miembros de ésta, pero ¿qué sucede con el objeto "cinturón"? En este caso concreto «POV» ya ha asignado una textura a "cinturón" y, por tanto no aplicará la "colorpiel". Resumiendo: en el caso de las texturas, la textura "global" sólo se aplicará sobre aquellos miembros de la unión que no tengan asignada su propia textura.

Pero, objetará el lector, lo que se ha asignado a "cinturón" es un pigmento y no una textura. ¡Pues no! Aunque no vamos a entrar ahora en demasiados detalles acerca de las sentencias de construcción de texturas, sí que diremos que las texturas "normales" de «POV» se definen con sentencias que se agrupan en tres apartados diferenciados: la pigmentación (pigment), las sentencias que perturban las normales (normal) y las que definen el acabado de la superficie (finish). Para cada uno de estos apartados «POV» tiene ya asignados unos valores por defecto (que pueden cambiarse), de manera que podemos ahorrarnos escribir la declaración completa de una textura cuando estamos creando una nueva. Así, en lugar de escribir algo como lo que aparece en la **Figura 3**...

...podemos limitarnos a escribir, por ejemplo:

pigment{ sentencias que afectan al color }

Pero cuando «POV» encuentra una línea como la anterior, lo que hace es aplicar una textura completa usando los valores por defecto para los otros apartados (lo mismo sucede con normal y finish). Por esta razón, en el ejemplo anterior la asignación del pigmento "MarrónClaro" provoca la asignación completa de una textura de manera que en el caso del objeto "cinturón" se ignora la textura "colorpiel".

Más simple aún es el caso de la sentencia siguiente del ejemplo de la **Figura 3**, donde tenemos una transformación espacial. Esta transformación se aplicará sin excepciones sobre todos los miembros de la unión. ¿Qué ocurre entonces, os preguntaréis, con el miembro head que tiene escrita una transformación dentro de su bloque? En este caso se aplicarán las dos. Primero «POV» efectuará la transformación particular de head, haciendo rotar la cabeza del orco y luego la unión completa (el orco) será trasladada a la posición resultante al aplicar la traslación global. (Y este será el resultado que obtendremos cuando la escena esté lista). Esta regla es idéntica para otros casos más complejos. Recordemos un ejemplo muy ilustrativo que ya fue incluido en el número 14 de Rendermanía:

```
...
...
texture{    pigment{ sentencias que afectan al color }
            normal{ sentencias que afectan a las normales }
            finish{ sentencias que afectan al acabado }
}
...
...
```

Figura 3

Fijaos en el caso del objeto manod (mano derecha). Sobre este objeto se aplican primero las transformaciones escritas dentro de la unión de la que forma parte. Después se le aplican las transformaciones de la unión de mayor nivel (el antebrazo) de la que forma parte la unión de la que es miembro. Y por último se efectúan las transformaciones de la unión principal que es el brazo. Este ejemplo –cuyo correcto funcionamiento quedó demostrado con los orcos del número 14 de Rendermanía– demuestra que «POV» trata las estructuras de bloques procesándolas de dentro a afuera. Esto puede ser empleado para simular en «POV» estructuras jerárquicas de objetos. En este ejemplo tenemos una estructura de este tipo donde la mano y el arma son objetos hijos del antebrazo, el cual, a su vez, es hijo del objeto que va unido al hombro. La utilidad de esto radica en que podemos dar una transformación dada a la mano pero ésta, además, obedecerá después primero a las transformaciones escritas para el antebrazo y luego a las señaladas para el brazo entero. Al llegar a este punto, quizá los pov-adeptos más novatos se sientan ya con fuerzas para afrontar el artículo dedicado a este tema en el Rendermanía 14 pero, como este caso es un poco complejo, os recomendamos que practiquéis con ejemplos más simples.

La sentencia unión es muy empleada también por las utilidades de traducción, que la utilizan para agrupar listas de triángulos de manera que se pueda asignar fácilmente texturas o transformaciones a los objetos formados por estas mallas de triángulos. También hay que decir que unión fue diseñada en principio como parte de la filosofía de modelado de «POV», la cual se apoya grandemente en las operaciones booleanas. Pero por ahora no vamos a entrar en ello porque dichas operaciones no funcionan con los objetos poligonales importados.

Algo sobre colores y texturas

Para poder crear una escena en «POV» diferente a una imagen absolutamente negra no basta con crear o cargar objetos en el pov-universo y encender una fuente de luz dentro de éste (además, claro está, de situar y orientar una cámara). También hay que asignar texturas a los objetos puesto que de lo contrario no podremos verlos aunque estén ante nuestras narices. Por ahora no vamos a estudiar las sentencias que nos permiten crear nuestras propias texturas y vamos a limitarnos a señalar que «POV» ya trae incorporada una gran librería de texturas para que podamos crear objetos con apariencia de madera, agua, piedra, cristal, metales, etc. Estas texturas predefinidas están almacenadas en los ficheros woods.inc, stones.inc, metals.inc, glass.inc y otros que encontraremos en el subdirectorio llamado "include", dentro del directorio donde tengamos a «POV». Estas texturas están creadas con #declare y asignadas a identificadores, por lo que para utilizarlas nos bastará con emplear la sentencia texture escribiendo dentro el nombre del identificador. En ciertos casos, no obstante, puede ser necesario algo más. Si por ejemplo estamos asignando una textura de madera a un objeto, los resultados dependerán en gran medida de la escala del objeto con respecto a la textura. Así, si el objeto resulta demasiado grande con respecto a la escala a la que se han definido los detalles de la textura, puede suceder que no percibamos dichos detalles al observar al objeto. Para remediar esto bastará con escribir una sentencia de escalación dentro del bloque de la textura. Así, por ejemplo, la sentencia:

```
object{head texture{T_Wood28 scale<2,2,2>}}
```

hace que la textura se aplique a una escala dos veces mayor de lo normal sobre el objeto head. Naturalmente esto es posible sólo porque, siguiendo las reglas del ámbito de los bloques, la operación de escala afecta sólo a la textura y no al objeto. Éste conservará el tamaño que tuviese pero los detalles de la textura pasarán a ser dos ve-

```
    ...
    ...
//brazo derecho
    union{
      union{object{antebd}
          object{munned}
//la mano y su arma cogen tambien el giro del antebrazo
          union{object{manod} object{arma}
              translate<39.5,-75,-2.5>
              rotate <rmanodX, rmanodY, rmanodZ>
              translate<-39.5,75,2.5>
          }
          translate<38,-104,-12>
          rotate <rantebdX, rantebdY, rantebdZ>
          translate<-38,104,12>
      }
      object{brazod}
      translate<25,-136,-9>
      rotate <rbrazodX, rbrazodY, rbrazodZ>
      translate<-25,136,9>
    }
    ...
    ...
```

Figura 4

ces más grandes. Quizá, observando la sentencia os preguntéis por qué hemos hecho que la escala se efectúe en los tres ejes y la respuesta es que T_Wood28 es una textura procedural normal de «POV» (definida en woods.inc con sentencias normales del lenguaje escénico, como las demás). Esto significa que, como todas las demás texturas algorítmicas de nuestro amado raytracer, ocupa un volumen infinito en todas direcciones.

Esto quizá parezca raro a los rendermaníacos que sólo hayan trabajado con las típicas texturas tipo bitmap, pero es así. En «POV» las texturas procedurales son como volúmenes tridimensionales que ocupan todo el universo espacial del programa. Cuando se declara un objeto dado en una posición espacial y se le asigna una de estas texturas, el objeto queda "impregnado" con el dibujo 3D que la textura tenga en esa posición. Para comprenderlo echad un vistazo a las dos escenitas de prueba. En la primera de ellas tenemos una esfera con una textura de madera tomada de woods.inc y en la siguiente tenemos a la misma esfera a la que se le ha cortado la mitad superior. El interior de la esfera también tiene la textura aplicada. (otros programas como «Imagine» o «3D Studio» también tienen sus propias texturas procedurales pero la gente casi nunca las usa, prefiriendo emplear bitmaps).

A estas alturas os habréis dado cuenta de que tendremos que escribir una sentencia #include para cargar el fichero woods.inc antes de referenciar a la textura T_Wood28. Si no lo hacemos así, «POV» nos dará un mensaje de error y la escena no se generará. (La práctica usual en estos casos es colocar todos los #includes al principio del fichero escénico). Pero además de esto hay que hacer una cosa más, hemos de indicarle a «POV» dónde están los ficheros. Por defecto, la orden #include busca los archivos que debe cargar en el directorio en curso pero puede suceder, como ocurre con los ficheros de texturas de «POV», que los archivos referenciados estén en otra parte. En estos casos «POV» seguirá buscando en los directorios que le hayamos indicado con el comando "L" de la línea de órdenes (más adelante veremos cómo funciona).

Por otro lado, «POV» también tiene definida una buena lista de colores que hallaréis declarados en colors.inc. Para usarlos únicamente tendremos que escribir el identificador del color dentro de la sentencia pigment. Pero,

En esta escena hemos empleado el mismo fichero que se utilizó para la imagen de portada de Rendermanía 14.

por supuesto, no tenemos por qué restringirnos al uso de esta lista de colores. Podemos crear nuestros propios colores usando sentencias como:

pigment{color rgb<1,0,0>}

o como:

pigment{rgb<1,0,0>}

En ambos casos se crea un pigmento rojo gracias a la sentencia RGB, la cual emplea valores que pueden oscilar de 0 a 1 para asignar intensidades a los componentes de rojo, verde y

Cada valor aleatorio genera una variación en la postura básica.

azul con los que se definen los colores en el pov-mundo. El valor 0 corresponde a la mínima intensidad y el 1 a la máxima posible. Los valores de estos componentes se complementan para producir todos los colores posibles. Así, por ejemplo, "RGB<1,0.48,0>" crea un color naranja. "RGB<1,1,1>" crea el color blanco, etc.

En fin, para acabar este resumen diremos que «POV» incluye algunos ficheros de ejemplo que muestran el resultado visual que ofrecen las texturas

El mismo valor de semilla genera idéntica postura para ambos personajes, como podéis apreciar en estas imágenes.

de su librería. Estos archivos están localizables en diversos subdirectorios del directorio texsamps (el cual, claro está, se halla dentro del directorio raíz donde hayáis metido «POV»).

Algunas órdenes de la línea de comandos

Para generar las imágenes correspondientes a los archivos mencionados antes o para crear alguna de las muchas escenas de ejemplo que incluye «POV», más valdrá que hablemos de las órdenes de línea más utilizadas que admite la versión MS-DOS de «POV». Estos comandos afectan a la forma en que trabaja «POV» y al fichero de salida con la escena que debe generarse. Por regla general suelen ir precedidos por un carácter + o -. El primero significa que la orden que le sigue se va a activar y el otro implica lo contrario. Los comandos deben escribirse en cualquier orden después de invocar al programa.

+I se usa para indicar el fichero escénico que «POV» debe procesar.

+O se emplea para indicar a «POV» el nombre que va a tener el archivo de salida.

+W indica la resolución horizontal de la imagen a generar.

+H indica la resolución vertical.

+D se usa para activar la visualización del modo gráfico mediante el cual podremos ir viendo como «POV» genera las imágenes línea a línea. Este comando acepta dos parámetros que se especifican con letras que deben colocarse juntas. La primera sirve para indicar el tipo de tarjeta gráfica disponible o para ordenar el empleo del estándar VESA (letra "G") y la segunda señala el tipo de paleta. Normalmente solemos utilizar las letras H (Hicolor) o T (Truecolor). A veces puede convenir desactivar la salida de vídeo, lo que haremos con -D.

+X Permite interrumpir la generación de la imagen pulsando una tecla.

+C Si interrumpimos la creación de una imagen podemos reanudar posteriormente el cálculo sin perder las líneas ya generadas usando +C.

+P Esta orden obliga a «POV» a esperar hasta que se pulse una tecla una vez que ha concluido la generación de la imagen. Se usa con +D para ver las imágenes.

```
...
...
camera {location <5900, 195, 0>
        direction <0.0, 0.0, 1>
        up  <0.0, 1.0, 0.0>
        right <.72,0.0, 0.0>
        look_at <0,70.5,0>
//  orthographic
}
...
...
```

Figura 5

+V Cuando hemos de desconectar la salida de vídeo es útil emplear +v para que «POV» vaya imprimiendo el número de línea con el que está trabajando en cada momento para hacernos una idea del tiempo que queda de render.

+A Activa el antialias. Normalmente suele añadirse un valor que indica la fuerza del antialias (el valor por defecto es 0). Esto se usa para suavizar la apariencia escalonada que tienen las líneas diagonales en la imagen.

+L Con esta opción podremos incluir una ruta al directorio donde tengamos ficheros .inc a los que vayamos a referenciar con #include desde el fichero escénico que se vaya a procesar.

+B Cada vez que se termina una línea «POV» hace una grabación a disco a menos que usemos +B para reducir el número de accesos al disco duro. Junto a +B añadiremos un valor que será el número de Kbytes que tendrá el buffer de grabación. La pega de esto es que si se va la luz o sucede un desastre habremos perdido todo lo que haya en el buffer.

Por último adjuntamos aquí la línea de órdenes que solemos utilizar (dentro de un fichero Bat).

povray +ibatalla.pov +obatalla.tga +dGH +x -v +p +w800 +h600 +a +b192 +lc:\pov302\include

Entre otras cosas, esta línea hará que «POV» genere el archivo batalla.tga a partir del fichero .pov del mismo nombre. Se activará la visualización usando el estándar VESA y Hicolor, se desactivará la estadística de líneas (con -v, esto es preciso si la visualización está activada), se creará un buffer de grabación de 192 Kbytes y se indica con +L una ruta para buscar los archivos .inc. También se emplea el antialias estándar.

La cámara

La cámara de POV es similar a la de otros programas de imagen sintética, es decir, una cámara no visible que POV emplea para tomar una foto del universo virtual descrito en el archivo escénico suministrado como entrada al programa. Por esta razón todos los ficheros escénicos deben incluir una descripción de una cámara válida utilizando las sentencias reservadas para ello en el lenguaje de POV. De lo contrario, y al igual que sucederá si no hay objetos o fuentes de luz definidas en la escena, no obtendremos otra cosa que una imagen con un fondo negro.

Por supuesto, podríamos evitarnos el estudio del manejo de la pov-cámara si nos limitásemos a emplear programas como «Rhino» o herramientas de traducción como 3ds2pov (sin trabajar la escena después desde POV), ya que, recordemos, estos programas graban en el fichero escénico exportado una cámara equivalente a la empleada desde «Rhino» o «3D Studio». Pero si verdaderamente deseamos utilizar las capacidades de POV nos veremos obligados a colocar la cámara desde el lenguaje escénico. Esta tarea puede ser, no nos engañemos, bastante trabajosa e ingrata comparada con el rápido manejo de la cámara que nos ofrece cualquier buen modelador. En efecto, a menudo nos resultará difícil predecir la imagen que se obtendrá de la colocación de una cámara determinada o el campo visual que abarcará ésta, y ello nos obligará a realizar bastantes pruebas.

El caballero emplea piezas intercambiables.

Las sentencias que definen la cámara suelen ir colocadas casi al principio del fichero escénico, después de los #includes, aunque pueden ser escritas en cualquier otra parte del archivo. En la **Figura 5** tenemos la cámara empleada para la escena de la portada.

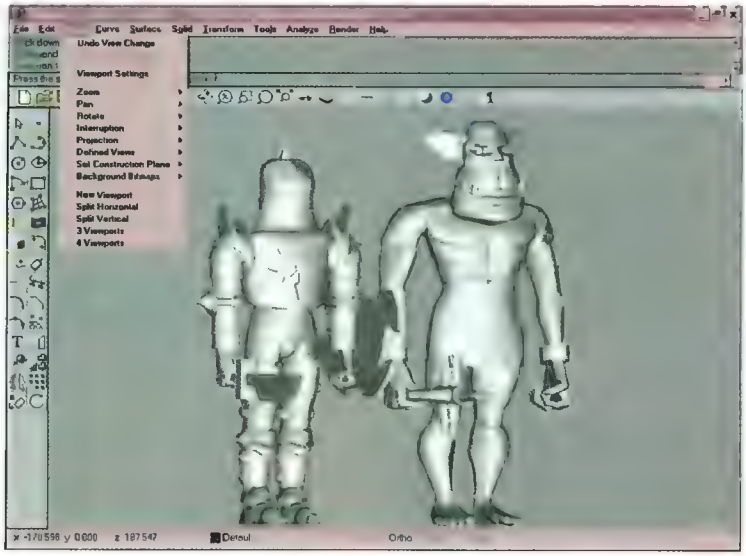
Las sentencias que definen a la cámara han de seguir a la sentencia "ca-

```
#include "colors.inc"
camera {
        location <0, 0, 0>
        direction <0,0 ,1>
        up  <0.0, 1.0, 0.0>
        right <1.33,0.0, 0.0>
}

light_source {<0,0,0> color White}

sphere{<0,0,5>,1 pigment{Green}}
sphere{<0,0,-5>,1 pigment{Blue}}
```

Figura 6



En la imagen se aprecia el momento de ajustar las dimensiones del caballero a las del orco.

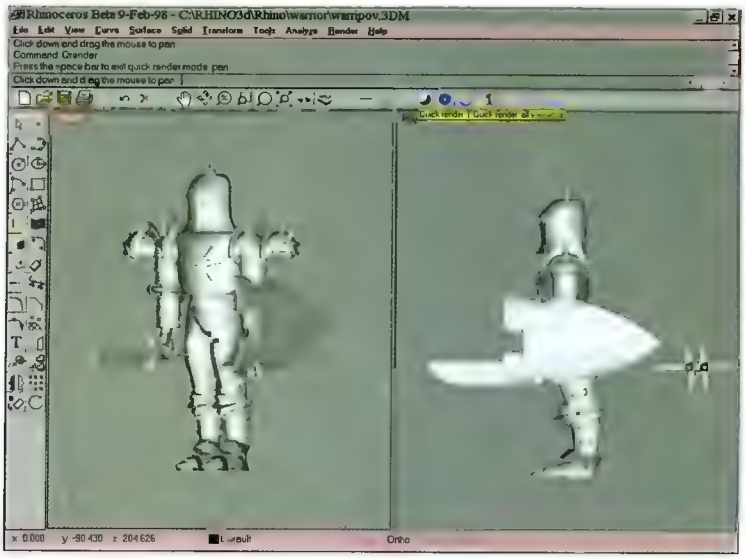
mera" y estar englobadas por corchetes, aunque no tienen por qué seguir el mismo orden que aparece en esta figura. La primera de ellas. "location". se utiliza para indicar –dando valores para los ejes X, Y y Z– la posición espacial de la cámara. Luego, para enfocar a ésta, deberemos emplear la sentencia "Look_at" (mirar a), con la que indicaremos el punto espacial hacia el que queremos apuntar la cámara. Al hallar esta sentencia. POV hará girar la cámara a la derecha o la izquierda y la inclinará hasta que el punto a enfocar quede apuntado por la cámara.

Llegados aquí se hace necesario explicar un par de puntos. La cámara no es un objeto sólido ni tiene dimensiones, pero debemos imaginar que, como en el caso de las cámaras reales, tiene una lente a la que llamaremos "ventana de visión". Esta ventana de visión tiene forma rectangular y sus características son controladas por las sentencias "direction", "up" y "right", las cuales controlan también el campo visual que abarca la mencionada ventana de visión. Direction define la dirección inicial hacia la que apunta la cámara. Así, por ejemplo, la sentencia direction "<0.0, 0.0, 1>" del ejemplo significa que la cámara inicialmente está mirando en la dirección +Z. El valor 1 también significa que la lente se encuentra a una unidad de distancia de la posición base de la cámara y esto determina –junto con el valor de up– el tamaño de la ventana de visión.

Para comprenderlo dibujad en una hoja un punto para representar la posición base de la cámara y también una línea vertical a la derecha del punto anterior (la línea ha de estar centrada verticalmente con respecto a dicho punto). Si ahora trazáis dos líneas que conecten el punto con los dos extremos de la línea vertical podréis visualizar el perfil de una cámara típica de POV, donde la apertura del campo de visión depende de la distancia de la ventana de visión (la línea) con respecto a la posición de la cámara (el punto). Observaréis que si arrastráis la línea vertical a la derecha las dos líneas diagonales (que representan el campo de visión de la cámara) formarán un ángulo menor entre sí. Traduciendo este ejemplo a POV, significa que un valor corto de direction implica un campo de visión mayor y un valor largo lo contrario. Fijaos en la **Figura 6**.

En el ejemplo no hay sentencia look_at. La cámara mira en la dirección +Z de manera que, al renderizar la escena, contemplaremos una esfera verde (green) en el centro de la escena. Aquí la lente de la cámara se encuentra a una unidad de distancia de la cámara. Si ahora cambiamos el 1 de direction por un 2 y volvemos a realizar la prueba la esfera será dos veces mayor. Esto se debe a que hemos alejado la ventana de visión de la cámara reduciendo el campo de visión, lo cual se aprecia como un zoom realizado sobre la dirección enfocada. (Si se sustituye el 1 por -1 la cámara apuntará a -Z y veremos la esfera azul).

En cuanto a las sentencias up y right, éstas controlan las proporciones de la ventana de visión, la orientación del sistema de coordenadas (right) y la dirección vertical de la ventana de visión (up). El valor 1.33 de right significa, por ejemplo, que la imagen va a ser un 1.33 más ancha que alta. Es decir, que vamos a ordenar con +w y +h una resolución como 640*480 ó 800*600 por ejemplo. En cambio, para generar una imagen de 1024 de alto por 768 de alto, el valor de right debería ser de .72. (Nota: raramente tocamos up y solamente cambiamos right cuando vamos a lanzar escenas con distintos formatos. Es peligroso cam-



Hemos creado un nuevo soldado con Rhinoceros.

biar right a negativo porque uno puede volverse loco si está acostumbrado al sistema "normal" de "mano izquierda" que utilizamos aquí. Casi siempre suele bastar con trastear con location y look_at. ¡Ojo! Poned siempre la sentencia look_at después de las demás ya mencionadas).

Tipos de cámara

En el primer ejemplo había una sentencia llamada "orthographic" anulada por un comentario. Esta sentencia cambiaba el tipo de perspectiva por defecto por otra en la que los rayos son paralelos. Este tipo de cámara sin perspectiva es muy útil cuando se quieren realizar pruebas de estimación de distancias o estamos estudiando las formas de un objeto y no queremos ser engañados por la perspectiva.

Aparte de orthographic existen otros tipos de cámara como fisheye (ojo de pez), omnimax, panoramic y cylindrical. Para comprobar su funcionamiento sólo tenéis que incluir la palabra correspondiente al final de las sentencias de la cámara. Y, por supuesto, también podemos crear nuestra propia cámara utilizando sentencias como angle.

Fuentes de luz

Para concluir con la pequeña introducción que estamos realizando sobre «POV» vamos a tratar brevemente el tema de la luz.

«POV» dispone de varios tipos de fuentes: ambientales, omnidireccionales, focos, cilíndricas y áreas de luz, y también cuenta con diversas posibilidades de control sobre estas fuentes. Sin embargo, nos limitaremos a hablar del tipo de fuente más sencilla y utilizada: la omnidireccional. El nombre de esta fuente se debe a que sus rayos parten en todas direcciones (como en el caso del sol o de una bombilla). Veamos su formato:

```
light_source {<0,0,0> color White}
```

Esta sentencia crea una fuente de luz omnidireccional de color blanco en las coordenadas <0,0,0>. El formato de la sentencia es, pues, muy sencillo. Primero incluimos la sentencia y dentro de los corchetes delimitadores colocamos, primero, las coordenadas de posición y luego el color de la fuente. Podemos indicar dicho color como en el caso anterior o empleando la sentencia "RGB" que ya vimos con los pigmentos:

```
light_source {<0,0,0> color rgb<1,1,1>}
```

Normalmente basta con una o dos fuentes de este tipo para iluminar cualquier escena –ninguna de las imágenes del presente número ha requerido más de dos fuentes–. Las fuentes de luz de «POV» son puntuales, es decir, todos los rayos que emiten parten del mismo punto. Esto implica dos cosas: la primera que no son visibles (aunque existe una manera de crear objetos que simulan ser luminosos) y la segunda que las sombras que generan tienen bordes excesivamente precisos y que por ello resultan bastante irreales (lo cual se puede solucionar utilizando áreas de luz).

Una buena iluminación es capital para lograr buenas e interesantes escenas. Si estáis trabajando sobre una imagen de tipo "a cielo abierto" lo mejor es situar las fuentes de luz a distancias astronómicas para que las sombras de todos los objetos tengan una orientación idéntica. También suele ser buena idea utilizar fuentes de colores distintos para crear algo de contraste y emplear áreas de luz si nos sobra tiempo (cada vez que sumemos una fuente a la escena estaremos aumentando el tiempo de cálculo necesario para generarla).

¡La batalla de la gran llanura!

El caballero se inclinó sobre la barandilla y miró hacia el suelo: a cientos de metros bajo sus pies se extendía un inmenso ejército. Arrugó la nariz y ordenó al oficial de navegación que pusiera rumbo al castillo. Con suerte el dirigible de exploración que mandaba llegaría al feudo a tiempo de prevenir al señor de aquella nueva invasión orca. Mientras el dirigible giraba lentamente, el caballero siguió observando con una sonrisa en los labios. ¡Sin duda se avecindaba una dura y estimulante batalla!

Una familia diferente genera una portada diferente.

Después de esperar durante meses el gran acontecimiento, estamos preparados para disfrutar de una batalla orco-humana digna de este nombre. Cientos de bestiales guerreros orcos han chocado contra las tropas humanas en un enfrentamiento largamente postergado. Ambos bandos han empleado algunas de las máquinas de guerra que enviasteis para apoyarles, y por ello la lucha ha sido verdaderamente feroz. A continuación os comentamos los detalles de este magno acontecimiento

El origen de la batalla

Todo empezó hace mucho tiempo con el número 1 de Rendermanía, cuando creamos unos lanceros bastante ridículos para poblar el castillo virtual que ilustró aquel número. En ese mismo número un autor publicó

un orco de aspecto bastante feroz y nos comentó que sin duda su criaturilla virtual podría hacer trizas sin problemas a los diminutos lanceros. Algo picados, decidimos reforzar a los lanceros con un cañón que apareció en el número 2 de Rendermanía, y a partir de aquí dio comienzo una inacabable carrera de armamentos.

Así, comenzasteis a enviar modelos de máquinas de guerra para apoyar a uno u otro bando y pronto la idea de crear una portada con una batalla medieval fantástica como tema surgió sin remedio. Se pretendía crear una escena con cientos de guerreros donde apareciesen las catapultas y las demás máquinas de guerra enviadas por los lectores.

Sin embargo, pasaron los meses y la batalla no parecía tomar forma en las páginas de Rendermanía. Finalmente, en el número 9 del suplemento, se publicó una portada donde aparecían tres parejas de contendientes (¡sólo 3!) peleándose sobre una superficie llena de libros. Esto fue decepcionante, por supuesto, pero al menos sirvió como base para estudiar los dos problemas principales que presentaba el diseño de este tipo de escenas:

• Los modelos eran mallas poligonales y cada personaje requería mucha RAM.

• No se disponía de un modelo articulado de orco; entonces se empleó un zombie en su lugar.

En el número 9 estos problemas no pudieron ser solucionados a tiempo pero no queriendo hacer esperar más a los lectores, se decidió emplear algunos de los modelos enviados para componer una portada. Al menos este número sirvió para comprobar lo sencillo que resultaba crear posturas de modelos con «Imagine» y cómo exportar modelos poligonales a «POV».

Sin embargo, pronto empezaron a resolverse los problemas que hacían imposible la ansiada batalla. Primero se creó un modelo articulable de orco en el número 11 empleando «Rhinoceros» y luego, en el número 12, se añadieron un par de cabezas más de orco. Finalmente descubrimos el uso de mesh y poco tiempo después esta sentencia fue empleada para crear el ejército de orcos que apareció en el número 14 de Rendermanía. A partir de aquí sólo era cuestión de tiempo preparar nuestra primera batalla virtual.

El bando humano

El primer paso una vez resueltos los problemas principales era contar con modelos adecuados para la batalla. El modelo del orco podía servir a pesar de sus fallos de articulación pero no ocurría lo mismo con el viejo lancero. Éste resultaba ya algo simplón comparado con el orco y se necesitaba otro modelo de aspecto más realista.

Por esta razón se ha creado un nuevo modelo de soldado que conserva, sin embargo, un cierto aire de familia con nuestro antiguo lancero, del que sólo hemos tomado las manos y el casco para el nuevo soldado.

En principio la idea era crear una armadura completa pero pronto se optó por crear también piezas internas para el cuerpo. De esta manera se podían eliminar trozos de armadura y crear una mayor variación. Así, en las escenas que ilustran el presente número, hay soldados con y sin protección en las piernas (lo que se controla desde los ficheros escénicos gracias a una variable). Esto queda bastante bien porque las piezas empleadas para el cuerpo del soldado han sido tomadas del cuerpo del orco. Naturalmente fue preciso editar grupos de vértices para lograr que el soldado resultase menos musculoso que el orco.

Pero la falta de tiempo hizo imposible llevar esta idea hasta sus últimas consecuencias y por ello nuestro soldado no tiene cabeza bajo el casco ni puede quitarse aún las protecciones de los brazos al no estar recortado el peto.

También hay que decir que nuestro soldado tiene fallos de articulación. Estos fallos son de difícil solución. Cuando, por ejemplo, el soldado levanta una pierna, ésta corta el faldón metálico de la armadura. Además, las hombreras pueden interseccionarse con el brazo si este se coloca en una posición excesivamente forzada, y lo mismo sucede con la placa del codo.

Al constatar esto, se probó, en primer lugar, que estos objetos rotasen acompañando el movimiento de las piezas de la armadura a que estaban unidas, pero el resultado era visualmente muy confuso, por lo que se optó por dejarlas sin movimiento.

Finalmente, con el fin de obtener algo de variedad, se crearon varias piezas adicionales: un segundo escudo, un par de cascos más y otra hombrera. De esta manera se confiaba en obtener soldados bastante diferentes entre sí. Pero, como siempre, la falta de tiempo impidió disponer de todas las piezas que hubiesen sido necesarias para que esta idea funcionase de verdad. Habría sido conveniente disponer de un par de petos más, de varias piezas intercambiables para brazos y piernas, de un faldón diferente, etc.

Preparación de las posturas

En principio la idea era tomar nota desde «Imagine» de los ángulos de rotación deseados para aplicarlos luego desde «POV», tal y como se hizo con las posturas creadas para el número 14. Pero pronto se comprobó que esto ya no era necesario.

En efecto, después de unas cuantas pruebas la experiencia hizo posible escoger a mano los valores medios de rotación para las posturas de correr, golpear, defender y estar muerto. De esta manera, tan sólo fue preciso escribir los ficheros .pov e .inc necesarios. Aquí se comprobó que, como el esquema de articulaciones de ambos modelos era similar, los dos podían compartir los mismos ficheros .inc de posturas, lo que permitió un significativo ahorro de trabajo.

Naturalmente esto fue posible porque se emplearon los mismos nombres para las variables de rotación en orco.inc y guerrero.inc. Por esta razón, ambos modelos pueden utilizar indistintamente los ficheros orcanda.inc, correr.inc, golpear.inc y otros.

Para quienes deseéis saber con más detalle cómo funciona esto os recomendamos leer el número 14 de Rendermanía, pues el proceso seguido este mes es el mismo. En resumen:

- Se crean los modelos desde un modelador poligonal cualquiera.
- Se toma nota desde el mismo modelador de las coordenadas de los ejes de rotación de cada pieza. Estos valores serán empleados luego en los ficheros .pov que definen la jerarquía de los modelos.
- Se graba cada pieza del modelo individualmente (exportándolas directamente a «POV» si el modelador lo permite o usando después una utilidad de traducción).
- Se escriben primero los ficheros de jerarquía (como orco.inc y guerrero.inc) y luego los que guardan los valores medios de rotación para obtener las posturas. (Estos son correr.inc, golpear.inc, etc). Los ficheros de jerarquía referencian a los ficheros .inc que guardan las mallas y hacen girar cada pieza o grupo de piezas según los valores de rotación suministrados por los archivos-postura. Además, los valores de rotación que se suministran a los archivos-jerarquía suelen ser siempre diferentes ya que los archivos-postura utilizan rangos aleatorios de libertad de movimiento para cada postura básica. De esta manera resulta posible obtener desde «POV» un número infinito de posturas a partir de un modelo almacenado en una única posición inicial.
- Después se crean los ficheros .pov que definen las escenas y que em-

plean bucles #while con cientos de pasadas sobre los ficheros con las posturas (primero) y los ficheros-jerarquía (después). Cada una de estas pasadas define un nuevo personaje en la escena.

Naturalmente todo esto ha sido posible gracias a las características de sentencias del lenguaje escénico como #declare, #include, #while, #if y otras. Y también al hecho de que cada objeto copia utiliza una ridícula cantidad de memoria en comparación a la que requiere el almacenamiento del modelo original al que se referencia con cada objeto-mesh.

Prueba de composición de la portada.

La batalla y otras ideas

Hemos preparado preferentemente escenas donde se ve a ambos ejércitos a punto de chocar. Aunque en principio solamente se iba a preparar una portada, pronto quedó claro que la idea daba para mucho más y se optó por crear muchas más escenas.

Esto resultaba tanto más preciso por cuanto era imposible aprovechar todos vuestros modelos en una única escena. De hecho, sólo hemos empleado algunos de vuestros primeros modelos, ya que no había tiempo de prepararlos a todos.

Así, hemos empleado el dirigible humano de Toni Sole, el magnífico tanque humano de vapor de Oscar Alvarez Díaz y la catapulta orca de Julián Hierro. Pero, ni que decir tiene que habrá nuevas contiendas entre orcos y humanos y para celebrarlas utilizaremos otros modelos diferentes entre los que nos habéis enviado.

Nota: nos hubiera gustado dar el mando de nuestros orcos a Ghorkas, el fantástico orco creado por José Luis Robledo Pescador. Pero la malla que envió no estaba entera –faltaban piezas en los brazos–. José Luis, si quieres que tu modelo aparezca en una próxima edición, por favor, envía una malla completa.

En cada nueva edición intentaremos preparar cosas nuevas. Y quizá algún día estemos preparados para crear, además, una descomunal animación con cientos de modelos corriendo o batallando entre sí.

Naturalmente todo esto también es aplicable a los mechs o a cualquier otro tema que se escoja como portada: últimamente se ha hablado bastante de preparar escenas espaciales.

En general se prepararan escenas de uno u otro tipo según el número de modelos que hayáis enviado. lo cual equivale también a enviar votos a favor de un tema dado para que estudiemos si es posible o no llevarlo a la práctica como protagonista de la portada.

Las escenas

Casi todo el presente número constituye una pequeña introducción a «POV», lo cual no guarda demasiada relación con las ilustraciones empleadas. El motivo de esto, aparte por supuesto de nuestras ganas de presentar por fin una contienda virtual, es que una introducción de este tipo sólo requiere unas escenas muy sencillas que, estamos seguros, los lectores no necesitarán para entender las explicaciones. En lugar de esto hemos preferido ilustrar Rendermanía con estas escenas para mostrar lo que «POV» es capaz de hacer.

NOTA: las escenas más complejas han requerido una media de 10 minutos de parsing (los ficheros .inc con las mallas tienen un tamaño de unos 50 Megas) y de 20 a 60 minutos para ser renderizadas sobre un Pentium a 150 MHz. ¿Quién dijo que «POV» es lento?

NOTA: Por falta de espacio, encontraréis dentro del fichero leedme.txt en el directorio Batalla dentro de la sección Rendermanía en el CD-ROM de la revista, las instrucciones oportunas para manejar los ficheros que adjuntamos.

Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en el sumario de Pcmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

Preguntas, sugerencias y polémicas

En este número, como siempre, nuestro foro es la galería de siempre donde podréis hallar magníficas escenas y alguna que otra animación. Sin embargo, este mes queremos hacer hincapié en el propio foro, es decir, en las cartas remitidas por los autores de estos trabajos. En ellas encontraréis descripciones de los procesos de creación, preguntas (a las que no siempre podemos contestar por falta de espacio), propuestas interesantes y opiniones diversas.

Este mes nuestra primera página del foro es para **Roberto Gracia**, el cual sólo ha tardado unas dos semanas en crear su propio trasatlántico virtual. Aunque claramente ambientado en el "Titanic", el "Oriental" ha sido diseñado enteramente por este autor, quien ha empleado «3D Studio 4» y «Spatch» para crear el monumental barco que podéis ver aquí. Sin embargo, el "Oriental" aún no está realmente listo para su botadura ya que, como cuenta el propio Roberto en su carta, aún faltan hamacas, sillas y otros detalles. Algo que Roberto ya tiene previsto hacer en la animación que espera preparar próximamente.

Hay que descubrirse ante el resultado obtenido por Roberto. El barco es muy convincente, tiene una estructura funcional dividida en cubiertas (conectadas entre sí con escaleras) y dispone también de los inevitables botes de salvamento. Roberto incluso ha utilizado un modelo humano para comprobar la correcta proporción de todos los detalles del "Oriental". En fin, un barco de verdadero lujo por el que te enviamos nuestras más sinceras felicitaciones. Confiamos que pronto podamos ver una animación en la que navegue por esos mares virtuales. (Nota: más detalles en la carta de Roberto, en el foro).

El Foro del Lector

Un grupo de rendermaniacos se ha unido para formar el grupo **ASTRO 3D** con el encomiable propósito de crear una librería de objetos de libre uso. En su carta, **Iñaki Karras** –uno de los miembros de este grupo– solicita la colaboración de otros rendermaniacos y da su dirección y la dirección e-mail de otro de los miembros. (Iñaki ha enviado también las dos escenas espaciales que podéis ver aquí). Lo dicho, una original iniciativa a la que deseamos el mayor de los éxitos.

Si queréis ver un buen combate espacial entre los inolvidables cazas rebeldes y los Tie imperiales echad un vistazo a la magnífica animación remitida por **Sergio Ruiz Navarro**. Tanto ésta como los modelos que aparecen en ella están pero que muy bien (sobre todo para alguien que lleva poco tiempo en esto). La coreografía de la batalla está muy lograda y las escenas donde se ve la cara del piloto aportan simpatía y realismo. Sin embargo, sentimos decir que el primer disco de la serie llegó mal y a causa de esto no hemos podido publicar los títulos.

En cuanto a tus preguntas. Para conocer las características del «MAX» te aconsejamos el libro publicado sobre este programa por Roberto Potenciano en Anaya. «POV», por otro lado, es un trazador de rayos que emplea un lenguaje para modelar en lugar de una interfaz gráfica como la empleada por «3D Studio». No olvidéis leer la carta de Sergio para poder ver la animación.

Francisco Javier Cruz Jiménez ha empleado a «Polyray» (el raytracer hermano de «POV») para crear su castillo y a «POV» y «Moray» para levantar la estupenda iglesia virtual que podéis ver aquí. Para la iglesia –que aún no está completamente acabada– Francisco ha usado también los famosos flares de Nathan Kopp. Sus cartas, donde describe el proceso de creación de ambos edificios, son una lectura recomendable para cualquier pov-adicto. Hay que destacar la forma en que Francisco creó la montaña y la integró con el castillo y el truco, sencillo pero efectivo, utilizado para ganar tiempo con las pruebas de render. En resumen, un buen trabajo.

Como ya hemos dicho más de una vez, la lectura de las cartas del foro suele ser una fuente de satisfacciones para nosotros, pero esto no siempre es así y como ejemplo de excepción presentamos hoy la carta de **Pedro J. Ladaria Linden**, el cual nos ha enviado varias escenas creadas con «3D Studio». La pena que hemos sentido al leer tu carta no se debe a la decepción que sufriste al ver la portada del número 9 (ya explicamos entonces por qué no pudimos hacer algo mejor) ni por tus opiniones sobre «POV», ni tampoco por tus escenas (que nos han gustado bastante y cuya crítica ya has hecho tú mismo). No, lo que nos ha fastidiado es el párrafo G de tu apartado de consejos: ¿Cómo puedes hablar así de la piratería? Es cierto que mucha gente tiene versiones piratas de programas de render. Esto no es algo especialmente dañino porque se trata de aficionados que no van a dedicarse profesionalmente a la infografía o que van a aprender antes de dedicarse legalmente a ella (y por tanto a comprar). Pero de comprender esto a recomendar la compra de CD piratas media un abismo. Nosotros hemos comprobado el daño que la piratería hace a la producción de software. Sabemos que...

1) No importa lo barato que pueda ser un programa. Existe un alto porcentaje de gente que siempre preferirá hacerse con una copia pirata. (¿Se sentirán más listos así?). 2) Hay muchos programadores de talento que se desaniman porque los beneficios de su duro trabajo no están garantizados por culpa de la piratería. Y no nos referimos aquí a programas caros sino a otros que están teóricamente al alcance de cualquier bolsillo y que bastante gente prefiere encontrar en alguno de esos CD piratas que mencionas. (Porque en ellos vienen muchos programas por el precio de uno. ¿No?)

Espero que recapacites sobre lo que has dicho y para terminar solo te diremos dos cosas:

¿Cuánto habrías tardado tú en renderizar una imagen como la del número 14, por ejemplo? Con «POV» una imagen de este tipo a 800*600 puede tardar de 30 a 45 minutos. La verdad es que sí sabemos algo de «MAX» pero las páginas de Rendermanía están dedicadas a programas share o free y a versiones libres de programas comerciales porque están al alcance de todos. ¿Y cuánto sabes tú de «POV» para hablar así de él?

Jordi y **Albert** son otro par de rendermaníacos adeptos a «3D Studio 4» y a «MAX». Nos ha gustado el anuncio de la pasta dentífrica y también las originales escenas con las poses futbolísticas. La cámara, la luz y las texturas son correctas pero creemos que aún deberíais ejercitaros más con el modelado.

Felipe Cañizares es un nuevo povmaníaco que tan solo llevaba dos semanas con «POV» cuando envío estas dos imágenes. Para él un par de consejos:

1) El libro que tienes no trata de la versión 3.0 pero empóllate como sea las sentencias de programación de POV. ¡Tus escenas ganarán mucho con ellas!

2) Procura emplear los efectos de «POV» sólo cuando tengas listos todos los detalles de la escena. Ganarás tiempo (y esto también va por las áreas de luz, aunque no sean ningún efecto).

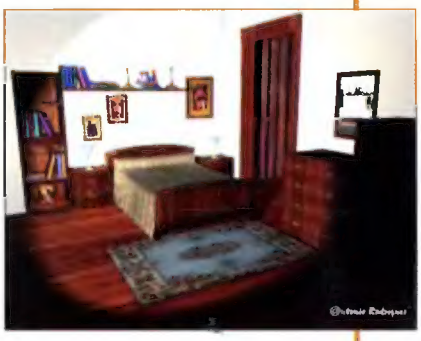
Tomo nota de tu sugerencia de las ciudades futuristas y las batallas espaciales para las portadas.

El Foro del Lector

De **Mario Prats Sánchez** nos llegan dos cartas. En la primera veremos al Arachno, un original mech en forma de araña preparado para todas las vicisitudes de la guerra virtual moderna. El modelo creado con Imagine y renderizado desde POV ha llegado demasiado tarde para la primera portada de mechs pero no temas. ¡Habrá mas! Arachno nos ha gustado mucho y tan solo cambiaría la forma de los pies y el valor de reflection, el cual nos parece alto. En cuanto a lo que dices de la memoria, no hay forma de saber a priori el tamaño que tendrá el .inc resultante de una conversión pero sí de saber la cantidad de RAM que usa «POV», ya que te lo dice el propio programa en la pantalla de estadísticas al acabar el render.

En su segunda carta (en el directorio segunda) Mario nos cuenta su paso a «MAX» y nos envía escenas con un castillo que parece de juguete. Esto se debe a la textura empleada y a la simplificación de algunos detalles. Por ello creemos que las escenas, resultonas gracias a las múltiples luces, habrían ganado si hubieras puesto el castillo sobre una mesa con otros juguetes, aunque esta idea tampoco es muy original.
Por último, pasaremos tu sugerencia sobre los plug-ins a redacción.

Antonio Rodríguez Gregores nos envía un estupendo Mech creado con «MAX» e «Imagine». Antonio ha enviado al Nayi 001-fx desglosado en partes para que nos sea más fácil prepararlo para «POV» con vistas a la portada de mechs (¡gracias, lo utilizaremos!) y también apunta una estupenda idea, que los propios lectores creen sus propias batallitas empleando mechs de otros autores. Así, en cada enfrentamiento, cada parte podría realizar una escena con su mech y el del contrincante, ganando el autor que hubiese preparado la escena más resultona. De momento Antonio ha desafiado ya a los hermanos Oscar y Alberto Otero Leal y a Benjamin Albares Moreno. ¿Alguien recoge el guante?



En cuanto al Nayi, se trata de un buen modelo al que quizá le falte un poco de detalle. Nos ha gustado más tu escena habit.jpg, que es una de las mejores habitaciones virtuales que han pasado por aquí (por la luz, las texturas, la disposición del cuarto, etc.)

Raúl Gómez Cardoso nos ha enviado unas simpáticas criaturillas virtuales creadas con «3D Studio 4». Por un lado está un orco-ogro y por otro una especie de alien-perro. Bien, no vamos a criticar tu obra pero te daremos un par de consejillos:
1) Hasta que no tengas más práctica en el modelado de formas orgánicas procura que tus modelos antropomórficos queden menos desnudos. Podrías, por ejemplo, haberle puesto una armadura al orco con lo cual habrías obviado la realización de ciertas partes difíciles. La verdad es que el cuerpo de tu modelo parece casi una armadura. Tan solo habrías tenido que cambiar el color verde del cuerpo y añadir algunas piezas más para que pareciese que el monstruo estaba dentro de una armadura. 2) Respecto a tus dificultades con «3D Studio 4» te recomendamos que te compres algún libro sobre el tema. Te ahorrarás muchos sufrimientos.